2

Core C#

WHAT'S IN THIS CHAPTER?

- Top-level statements
- Declaring variables
- Target-typed new expressions
- Nullable types
- Predefined C# data types
- Program flow
- Namespaces
- Strings
- Comments and documentation
- C# preprocessor directives
- Guidelines and coding conventions

CODE DOWNLOADS FOR THIS CHAPTER

The source code for this chapter is available on the book page at `www.wiley.com`. Click the Downloads link. The code can also be found at `https://github.com/ProfessionalCSharp/ProfessionalCSharp2021` in the directory `1_CS/CoreCSharp`.

The code for this chapter is divided into the following major examples:

- TopLevelStatements
- CommandLineArgs
- VariableScopeSample
- NullableValueTypes

- NullableReferenceTypes
- ProgramFlow
- SwitchStatement
- SwitchExpression
- ForLoop
- StringSample

All the sample projects have nullable reference types enabled.

FUNDAMENTALS OF C#

Now that you understand more about what C# can do, you need to know how to use it. This chapter gives you a good start in that direction by providing a basic understanding of the fundamentals of C# programming, which subsequent chapters build on. By the end of this chapter, you will know enough C# to write simple programs (though without using inheritance or other object-oriented features, which are covered in later chapters).

The previous chapter explained how to create a "Hello, World!" application using the .NET CLI tools. This chapter focuses on C# syntax. First, here's some general information on the syntax:

- Statements end in a semicolon (`;`) and can continue over multiple lines without needing a continuation character.
- Statements can be joined into blocks using curly braces (`{}`).
- Single-line comments begin with two forward slash characters (`//`).
- Multiline comments begin with a slash and an asterisk (`/*`) and end with the same combination reversed (`*/`).
- C# is case-sensitive. Variables named `myVar` and `MyVar` are two different variables.

Top-Level Statements

A new feature of C# 9 is top-level statements. You can create simple applications without defining a namespace, declaring a class, and defining a `Main` method. A one-line "Hello, World!" application can look like this:

```
System.Console.WriteLine("Hello World!");
```

Let's enhance this one-line application to open the namespace where the `Console` class is defined first. With the `using` directive to import the `System` namespace, you can use class `Console` without prefixing it with the namespace:

```
using System;
Console.WriteLine("Hello World!");
```

Because `WriteLine` is a static method of the `Console` class, it's even possible to open the `Console` class with the `using static` directive:

```
using static System.Console;
WriteLine("Hello World!");
```

Behind the scenes, with top-level statements, the compiler creates a class with a `Main` method and adds the top-level statements to the `Main` method:

```
using System;

class Program
{
```

```
  static void Main()
  {
    Console.WriteLine("Hello, World!");
  }
}
```

> **NOTE** *Many of the samples of this book use top-level statements because this feature is extremely useful with small sample applications. This feature can also be of practical use with small microservices that you now can write in a few code lines and when you use C# in a scripting-like environment.*

Variables

C# offers different ways to declare and initialize variables. A variable has a type and a value that can change over time. In the next code snippet, the variable `s1` is of type `string` as defined with the type declaration at the left of the variable name, and it is initialized to a new string object where the string literal `"Hello, World!"` is passed to the constructor. Because the `string` type is commonly used, instead of creating a new string object, the string `"Hello, World!"` can be directly assigned to the variable (shown with the variable `s2`).

C# 3 invented the `var` keyword with *type inference*, which can be used to declare a variable as well. Here, the type is required on the right side, and the left side would infer the type from it. As the compiler creates a string object from the string literal `"Hello, World"`, `s3` is in the same way a type-safe strongly defined string like `s1` and `s2`.

C# 9 provides another new syntax to declare and initialize a variable with the *target-typed new expression*. Instead of writing the expression `new string("Hello, World!")`, if the type is known at the left side, using just the expression `new("Hello, World!")` is sufficient; you don't have to specify the type on the right side (code file `TopLevelStatements/Program.cs`):

```
using System;

string s1 = new string("Hello, World!");
string s2 = "Hello, World!";
var s3 = "Hello, World!";
string s4 = new("Hello, World!");

Console.WriteLine(s1);
Console.WriteLine(s2);
Console.WriteLine(s3);
Console.WriteLine(s4);
//...
```

> **NOTE** *Declaring the type on the left side using the `var` keyword or the target-typed new expression often is just a matter of taste. Behind the scenes, the same code gets generated. The `var` keyword has been available since C# 3 and reduced the amount of code you needed to write by defining the type both on the left side to declare the type and on the right side when instantiating the object. With the `var` keyword, you only have to have the type on the right side. However, the `var` keyword cannot be used with members of types. Before C# 9, you had to write the type two times with class members; now you can use target-typed new. Target-typed new can be used with local variables, which you can see in the preceding code snippet with variable `s4`. This doesn't make the `var` keyword useless; it still has its advantages—for example, on receiving values from a method.*

Command-Line Arguments

When you're passing values to the application when starting the program, the variable `args` is automatically declared with top-level statements. In the following code snippet, with the `foreach` statement, the variable `args` is accessed to iterate through all the command-line arguments and display the values on the console (code file `CommandLineArgs/Program.cs`):

```
using System;

foreach (var arg in args)
{
  Console.WriteLine(arg);
}
```

Using the .NET CLI to run the application, you can use `dotnet run` followed by `--` and then pass the arguments to the program. The `--` needs to be added so as not to confuse the arguments of the .NET CLI with the arguments of the application:

```
> dotnet run -- one two three
```

When you run this, you see the strings `one two three` on the console.

When you create a custom `Main` method, the method needs to be declared to receive a string array. You can choose a name for the variable, but the variable named `args` is commonly used, which is the reason this name was selected for the automatically generated variable with top-level statements:

```
using System;

class Program
{
  static void Main(string[] args)
  {
    foreach (var arg in args)
    {
      Console.WriteLine(arg);
    }
  }
}
```

Understanding Variable Scope

The *scope* of a variable is the region of code from which the variable can be accessed. In general, the scope is determined by the following rules:

- A *field* (also known as a member variable) of a class is in scope for as long as its containing class is in scope.
- A *local variable* is in scope until a closing brace indicates the end of the block statement or method in which it was declared.
- A local variable that is declared in a `for`, `while`, or similar statement is in scope in the body of that loop.

It's common in a large program to use the same variable name for different variables in different parts of the program. This is fine as long as the variables are scoped to completely different parts of the program so that there is no possibility for ambiguity. However, bear in mind that local variables with the same name can't be declared twice in the same scope. For example, you can't do this:

```
int x = 20;
// some more code
int x = 30;
```

Consider the following code sample (code file `VariableScopeSample/Program.cs`):

```
using System;

for (int i = 0; i < 10; i++)
{
  Console.WriteLine(i);
} // i goes out of scope here

// We can declare a variable named i again, because
// there's no other variable with that name in scope
for (int i = 9; i >= 0; i--)
{
  Console.WriteLine(i);
} // i goes out of scope here.
```

This code simply prints out the numbers from 0 to 9, and then from 9 to 0, using two `for` loops. The important thing to note is that you declare the variable `i` twice in this code, within the same method. You can do this because `i` is declared in two separate loops, so each `i` variable is local to its own loop.

Here's another example (code file `VariableScopeSample2/Program.cs`):

```
int j = 20;
for (int i = 0; i < 10; i++)
{
  int j = 30; // Can't do this – j is still in scope
  Console.WriteLine(j + i);
}
```

If you try to compile this, you get an error like the following:

```
error CS0136: A local or parameter named 'j' cannot be declared in this scope because
that name is used in an enclosing local scope to define a local or
parameter
```

This occurs because the variable `j`, which is defined before the start of the `for` loop, is still in scope within the `for` loop and won't go out of scope until the `Main` method (which is created from the compiler) has finished executing. The compiler has no way to distinguish between these two variables, so it won't allow the second one to be declared.

It even doesn't help to put the variable `j` declared outside of the `for` loop after the end of the `for` loop. The compiler moves all variable declarations at the beginning of a scope no matter where you declare it.

Constants

For values that never change, you can define a constant. For constant values, you can use the `const` keyword.

With variables declared with the `const` keyword, the compiler replaces the variable in every occurrence with the value specified with the constant.

A constant is specified with the `const` keyword before the type:

```
const int a = 100; // This value cannot be changed.
```

The compiler replaces every occurrence of the local field with the value. This behavior is important in terms of versioning. If you declare a constant with a library and use the constant from an application, the application needs to be recompiled to get the new value; otherwise, the library could have a different value from the application. Because of this, it's best to use `const` only with values that never change, even in future versions.

Constants have the following characteristics:

- They must be initialized when they are declared. After a value has been assigned, it can never be overwritten.
- The value of a constant must be computable at compile time. You can't initialize a constant with a value taken from a variable. If you need to do this, you must use a read-only field.
- Constants are always implicitly static. Notice that you don't have to (and, in fact, are not permitted to) include the `static` modifier in the constant declaration.

The following are the advantages of using constants in your programs:

- Constants make your programs easier to read by replacing magic numbers and strings with readable names whose values are easy to understand.
- Constants help prevent mistakes in your programs. If you attempt to assign another value to a constant somewhere in your program other than at the point where the constant is declared, the compiler flags the error.

> **NOTE** *If multiple instances could have different values but the value never changes after initialization, you can use the* `readonly` *keyword. This is discussed in Chapter 3, "Classes, Records, Structs, and Tuples."*

Methods and Types with Top-Level Statements

You can also add methods and types to the same file with top-level statements. In the following code snippet, the method named `Method` is defined and invoked after the method declaration and implementation (code file `TopLevelStatements/Program.cs`):

```
//...
void Method()
{
    Console.WriteLine("this is a method");
}

Method();
//...
```

The method can be declared before or after it is used. Types can be added to the same file, but these need to be specified following the top-level statements. With the following code snippet, the class `Book` is specified to contain a `Title` property and the `ToString` method. Before the declaration of the type, a new instance is created and assigned to the variable `b1`, the value of the `Title` property is set, and the instance is written to the console. When the object is passed as an argument to the `WriteLine` method, in turn the `ToString` method of `Book` class is invoked:

```
Book b1 = new();
b1.Title = "Professional C#";
Console.WriteLine(b1);


class Book
{
```

```
  public string Title { get; set; }
  public override string ToString() => Title;
}
```

> **NOTE** *Creating and invoking methods and defining classes are explained in detail in Chapter 3.*

> **NOTE** *All the top-level statements need to reside in one file. Otherwise, the compiler wouldn't know where to start. If you use top-level statements, make them easy to find, such as by adding them to the* `Program.cs` *file. You don't want to search for the top-level statements in a list of multiple files.*

NULLABLE TYPES

With the first version of C#, a value type couldn't have a null value, but it was always possible to assign null to a reference type. The first change happened with C# 2 and the invention of the *nullable value type*. C# 8 brought a change with reference types because most exceptions occurring with .NET are of type `NullReferenceException`. These exceptions occur when a member of a reference is invoked that has `null` assigned. To reduce these issues and get compiler errors instead, *nullable reference types* were introduced with C# 8.

This section covers both nullable value types and nullable reference types. The syntax looks similar, but it's very different behind the scenes.

Nullable Value Types

With a value type such as `int`, you cannot assign `null` to it. This can lead to difficulties when mapping to databases or other data sources, such as XML or JSON. Using a reference type instead results in additional overhead: an object is stored in the heap, and the garbage collection needs to clean it up when it's not used anymore. Instead, the `?` can be used with the type definition, which allows assigning `null`:

```
int? x1 = null;
```

The compiler changes this to use the `Nullable<T>` type:

```
Nullable<int> x1 = null;
```

`Nullable<T>` doesn't add the overhead of a reference type. This is still a `struct` (a value type) but adds a Boolean flag to specify if the value is `null`.

The following code snippet demonstrates using nullable value types and assigning non-nullable values. The variable `n1` is a nullable `int` that has been assigned the value `null`. A nullable value type defines the property `HasValue`, which can be used to check whether the variable has a value assigned. With the `Value` property, you can access its value. This can be used to assign the value to a non-nullable value type. A non-nullable value can always be assigned to a nullable value type; this always succeeds (code file `NullableValueTypes/Program.cs`):

```
int? n1 = null;
if (n1.HasValue)
{
  int n2 = n1.Value;
}
int n3 = 42;
int? n4 = n3;
```

Nullable Reference Types

Nullable reference types have the goal of reducing exceptions of type `NullReferenceException`, which is the most common exception that occurs with .NET applications. There always has been a guideline that an application should not throw such exceptions and should always check for `null`, but without the help of the compiler, such issues can be missed too easily.

To get help from the compiler, you need to turn on nullable reference types. Because this feature has breaking changes with existing code, you need to turn it on explicitly. You specify the `Nullable` element and set the `enable` value in the project file (project file `NullableReferenceTypes.csproj`):

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>
</Project>
```

Now, `null` cannot be assigned to reference types. When you write this code with nullable enabled,

```
string s1 = null; // compiler warning
```

you get the compiler warning "CS8600: Converting a null literal or a possible null value to non-nullable type."

To assign null to the string, the type needs to be declared with a question mark—like nullable value types:

```
string? s1 = null;
```

When you're using the nullable `s1` variable, you need to make sure to verify for not `null` before invoking methods or assigning it to non-nullable strings; otherwise, compiler warnings are generated:

```
string s2 = s1.ToUpper(); // compiler warning
```

Instead, you can check for `null` before invoking the method with the *null-conditional operator* `?.`, which invokes the method only if the object is not `null`. The result cannot be written to a non-nullable string. The result of the right expression can be `null` if `s1` is `null`:

```
string? s2 = s1?.ToUpper();
```

You can use the *coalescing operator* `??` to define a different return value in the case of `null`. With the following code snippet, an empty string is returned in case the expression to the left of `??` returns `null`. The complete result of the right expression is now written to the variable `s3`, which can never be `null`. It's either the uppercase version of the `s1` string if `s1` is not `null`, or an empty string if `s1` is `null`:

```
string s3 = s1?.ToUpper() ?? string.Empty;
```

Instead of using these operators, you can also use the `if` statement to verify whether a variable is not `null`. With the `if` statement in the following code snippet, the C# pattern `is not` is used to verify that `s1` is not `null`. The block covered by the `if` statement is invoked only when `s1` is not `null`. Here it is not necessary to use the null-conditional operator to invoke the method `ToUpper`:

```
if (s1 is not null)
{
  string s4 = s1.ToUpper();
}
```

Of course, it's also possible to use the *not equals operator* `!=`:

```
if (s1 != null)
{
  string s5 = s1.ToUpper();
}
```

> **NOTE** *Operators are covered in detail in Chapter 5, "Operators and Casts."*

Using nullable reference types is also important with members of types, as shown in the `Book` class with the `Title` and `Publisher` properties in the following code snippet. The `Title` is declared with a non-nullable `string` type; thus, it needs to be initialized when creating a new object of the `Book` class. It's initialized with the constructor of the `Book` class. The `Publisher` property is allowed to be `null`, so it doesn't need initialization (code file `NullableReferenceTypes/Program.cs`):

```
class Book
{
  public Book(string title) => Title = title;

  public string Title { get; set; }
  public string? Publisher { get; set; }
}
```

When you're declaring a variable of the `Book` class, the variable can be declared as nullable (`b1`), or it needs a `Book` object with the declaration using the constructor (`b2`). The `Title` property can be assigned to a non-nullable `string` type. With the `Publisher` property, you can assign it to a nullable string or use the operators as shown earlier:

```
Book? b1 = null;
Book b2 = new Book("Professional C#");
string title = b2.Title;
string? publisher = b2.Publisher;
```

Behind the scenes with nullable value types, the type `Nullable<T>` is used behind the scenes. This is not the case with nullable reference types. Instead, the compiler adds annotation to the types. Nullable reference types have `Nullable` attributes associated. With this, nullable reference types can be used with libraries to annotate parameters and members with nullability. When the library is used with new applications, IntelliSense can give information regarding whether a method or property can be `null`, and the compiler acts accordingly with compiler warnings. Using an older version of the compiler (earlier than C# 8), the library can still be used in the same way nonannotated libraries are used. The compiler just ignores the attributes it doesn't know.

> **NOTE** *Nearly all the samples of this book are configured to have nullable reference types turned on. With .NET 5, nearly all the base class libraries have been fully annotated with nullable reference types. This helps getting information about what is required with parameters and what is returned. An interesting aspect here is the choices Microsoft made in deciding nullability. The string returned from the* `object.ToString` *method was originally documented that overriding this method should never return null. The .NET team reviewed different implementations: some Microsoft teams overriding this method returned null. Because the usage was different than the documentation, Microsoft decided to declare the* `object.ToString` *method to return* `string?`*, which allows it to return* `null`*. Overriding this method, you can be stricter and return* `string`*. Overriding methods is explained in detail in Chapter 4, "Object-Oriented Programming in C#."*
>
> *Because nullable reference types is a breaking change when turning this feature on with existing applications, to allow for a slow migration to this new feature, you can use the preprocessor directive* `#nullable` *to turn it on or off and to restore it to the setting from the project file. This is discussed in the section "C# Preprocessor Directives."*

USING PREDEFINED TYPES

Now that you have seen how to declare variables and constants and know about an extremely important enhancement with nullability, let's take a closer look at the data types available in C#.

The C# keywords for data types—such as `int`, `short`, and `string`—are mapped from the compiler to .NET data types. For example, when you declare an `int` in C#, you are actually declaring an instance of a .NET struct: `System.Int32`. All the primitive data types offer methods that can be invoked. For example, to convert `int i` to a `string`, you can write the following:

```
string s = i.ToString();
```

I should emphasize that behind this syntactical convenience, the types really are stored as primitive types, so absolutely no performance cost is associated with the idea that the primitive types are represented by .NET structs.

The following sections review the types that are recognized as built-in types in C#. Each type is listed along with its definition and the name of the corresponding .NET type. I also show you a few exceptions—some important data types that are available only with their .NET type and don't have a specific C# keyword.

Let's start with predefined value types that represent primitives, such as integers, floating-point numbers, characters, and Booleans.

Integer Types

C# supports integer types with various numbers of bits used and differs between types that support only positive values or types with a range of negative and positive values. Eight bits are used by the `byte` and `sbyte` types. The `byte` type allows values from 0 to 255—only positive values—whereas the `s` in `sbyte` means to use a sign; that type supports values from –128 to 127, which is what's possible with 8 bits.

The `short` and `ushort` types make use of 16 bits. The `short` type covers the range from –32,768 to 32,767. With the `ushort` type, the `u` is for unsigned, and it covers 0 to 65,535. Similarly, the `int` type is a signed 32-bit integer, and the `uint` type is an unsigned 32-bit integer. `long` and `ulong` have 64 bits available. Behind the scenes, the C# keywords `sbyte`, `short`, `int`, and `long` map to `System.SByte`, `System.Int16`, `System.Int32`, and `System.Int64`. The unsigned versions map to `System.Byte`, `System.UInt16`, `System.UInt32`, and `System.UInt64`. The underlying .NET types clearly list the number of bits used in the name of the type.

To check for the maximum and minimum values from the type, you can use the `MaxValue` and `MinValue` properties.

Big Integer

In case you need a number representation that has a bigger value than the 64 bits available in the `long` type, you can use the `BigInteger` type. This struct doesn't have a limit on the number of bits and can grow until there's not enough memory available. There's not a specific C# keyword for this type, and you need to use `BigInteger`. Because this type can grow endlessly, `MinValue` and `MaxValue` properties are not available. This type offers built-in methods for calculation such as `Add`, `Subtract`, `Divide`, `Multiply`, `Log`, `Log10`, `Pow`, and others.

Native Integer Types

With `int`, `short`, and `long`, the number of bits and available sizes are independent if the application is a 32- or 64-bit application. This is different from the integer definitions as defined with C++. C# 9 has new keywords for platform-specific values: `nint` and `nuint` (native integer and native unsigned integer, respectively). In a 64-bit application, these integer types make use of 64 bits, whereas in a 32-bit application just 32 bits are used. These types are important with direct memory access, which is covered in Chapter 13, "Managed and Unmanaged Memory."

Digit Separators

For better readability of numbers, you can use digit separators. You can add underscores to numbers, as shown in the following code snippet. In this code snippet, also the `0x` prefix is used to specify hexadecimal values (code file `DataTypes/Program.cs`):

```
long l1 = 0x_123_4567_89ab_cedf;
```

The underscores used as separators are just ignored by the compiler. These separators help with readability and don't add any functionality. With the preceding sample, reading from the right, every 16 bits (or 4 hexadecimal characters) a digit separator is added. This is a lot more readable compared to this:

```
long l2 = 0x123456789abcedf;
```

Of course, because the compiler ignores the underscores, you are responsible for readability yourself. You can put the underscores at any position, which may not really help with readability:

```
long l3 = 0x_12345_6789_abc_ed_f;
```

It's useful that any position can be used, which allows for different use cases such as to work with hexadecimal or octal values or to separate different bits needed for a protocol, as shown in the next section.

Binary Values

Besides offering digit separators, C# also makes it easy to assign binary values to integer types. Using the `0b` literal, it's only allowed to assign values of 0 and 1, such as the following (code file `DataTypes/Program.cs`):

```
uint binary1 = 0b_1111_1110_1101_1100_1011_1010_1001_1000;
```

The preceding code snippet uses an unsigned int with 32 bits available. Digit separators help with readability for using binary values. This snippet makes a separation every 4 bits. Remember, you can write this in the hex notation as well:

```
uint hex1 = 0xfedcba98;
```

Using the separator every 3 bits helps in working with the octal notation, where characters are used between 0 (000 binary) and 7 (111 binary).

```
uint binary2 = 0b_111_110_101_100_011_010_001_000;
```

If you need to define a binary protocol—for example, where 2 bits define the rightmost part followed by 6 bits in the next section, and two times 4 bits to complete 16 bits—you can put separators per this protocol:

```
ushort binary3 = 0b1111_0000_101010_11;
```

> **NOTE** *Read Chapter 5 for additional information on working with binary data.*

Floating-Point Types

C# also specifies floating-point types with different numbers of bits based on the IEEE 754 standard. The `Half` type (new as of .NET 5) uses 16 bits, `float` (`Single` with .NET) uses 32 bits, and `double` (`Double`) uses 64 bits. With all of these data types, 1 bit is used for the sign. Depending on the type, 10 through 52 bits are used for the significand, and 5 through 11 bits for the exponent. The following table shows the details:

C# KEYWORD	.NET TYPE	DESCRIPTION	SIGNIFICAND BIT	EXPONENT BIT
	`System.Half`	16-bit, single-precision floating point	10	5
`float`	`System.Single`	32-bit, single-precision floating point	23	8
`double`	`System.Double`	64-bit, double-precision floating point	52	11

When you assign a value, if you hard-code a noninteger number (such as 12.3), the compiler assumes that's a double. To specify that the value is a `float`, append the character `F` (or `f`):

```
float f = 12.3F;
```

With the `decimal` type (.NET struct `Decimal`), .NET has a high-precision floating-point type that uses 128 bits and can be used for financial calculations. With the 128 bits, 1 is used for the sign, and 96 for the integer number. The remaining bits specify a scaling factor. To specify that your number is a `decimal` type rather than a `double`, a `float`, or an integer, you can append the `M` (or `m`) character to the value:

```
decimal d = 12.30M;
```

The Boolean Type

You use the C# `bool` type to contain Boolean values of either `true` or `false`.

You cannot implicitly convert `bool` values to and from integer values. If a variable (or a function return type) is declared as a `bool`, you can only use values of `true` and `false`. You get an error if you try to use zero for `false` and a nonzero value for `true`.

The Character Type

The .NET string consists of two-byte characters. The C# keyword `char` maps to the .NET type `Char`. Using single quotation marks, for example, `'A'`, creates a char. With double quotation marks, a string is created.

As well as representing chars as character literals, you can represent them with four-digit hex Unicode values (for example, `'\u0041'`), as integer values with a cast (for example, `(char)65`), or as hexadecimal values (for example,`'\x0041'`). You can also represent them with an escape sequence, as shown in the following table:

ESCAPE SEQUENCE	CHARACTER
`\'`	Single quotation mark
`\"`	Double quotation mark
`\\`	Backslash
`\0`	Null
`\a`	Alert
`\b`	Backspace
`\f`	Form feed

ESCAPE SEQUENCE	CHARACTER
`\n`	Newline
`\r`	Carriage return
`\t`	Tab character
`\v`	Vertical tab

Literals for Numbers

In the preceding sections, literals have been shown for numeric values. Let's summarize them here in the following table:

LITERAL	POSITION	DESCRIPTION
`U`	Postfix	`unsigned int`
`L`	Postfix	`long`
`UL`	Postfix	`unsigned long`
`F`	Postfix	`float`
`M`	Postfix	`decimal` (money)
`0x`	Prefix	Hexadecimal number; values from `0` to `F` are allowed
`0b`	Prefix	Binary number; only `0` and `1` are allowed
`true`	NA	Boolean value
`false`	NA	Boolean value

The object Type

Besides value types, with C# keywords, two reference types are defined: the `object` keyword that maps to the `Object` class and the `string` keyword that maps to the `String` class. The `string` type is discussed later in this chapter in the section "Working with Strings." The `Object` class is the ultimate base class of all reference types and can be used for two purposes:

- You can use an `object` reference to bind to an object of any particular subtype. For example, in Chapter 5, you'll see how you can use the `object` type to box a value object on the stack to move it to the heap; `object` references are also useful in reflection, when code must manipulate objects whose specific types are unknown.
- The `object` type implements a number of basic, general-purpose methods, which include `Equals`, `GetHashCode`, `GetType`, and `ToString`. User-defined classes might need to provide replacement implementations of some of these methods using an object-oriented technique known as *overriding*, which is discussed in Chapter 4. When you override `ToString`, for example, you equip your class with a method for intelligently providing a string representation of itself. If you don't provide your own implementations for these methods in your classes, the compiler picks up the implementations of the `object` type, which returns the name of the class.

CONTROLLING PROGRAM FLOW

This section looks at the real nuts and bolts of the language: the statements that allow you to control the *flow* of your program rather than execute every line of code in the order it appears in the program. With conditional statements like the `if` and `switch` statements, you can branch your code depending on whether certain conditions are met. You can repeat statements in loops with `for`, `while`, and `foreach` statements.

The if Statement

With the `if` statement, you can specify an expression within parentheses. If the expression returns `true`, the block that's specified with curly braces is invoked. In case the condition is not `true`, you can check for another condition to be true using `else if`. The `else if` can be repeated to check for more conditions. If neither the expressions specified with the `if` nor all the `else if` expressions evaluate to `true`, the block specified with the `else` block is invoked.

With the following code snippet, a string is read from the console. If an empty string is entered, the code block following the `if` statement is invoked. The `string` method `IsNullOrEmpty` returns `true` if the `string` is either `null` or empty. The block specified with the `else if` statement is invoked when the length of the input is smaller than five characters. In all other cases—for example, with an input length of five or more characters—the `else` block is invoked (code file `ProgramFlow/Program.cs`):

```
Console.WriteLine("Type in a string");
string? input = Console.ReadLine();

if (string.IsNullOrEmpty(input))
{
  Console.WriteLine("You typed in an empty string.");
}
else if (input?.Length < 5)
{
  Console.WriteLine("The string had less than 5 characters.");
}
else
{
  Console.WriteLine("Read any other string");
}
Console.WriteLine("The string was " + input);
```

> **NOTE** *If there's just a single statement with the* `if`/`else if`/`else` *blocks, the curly braces are not necessary. They are necessary only with multiple statements. However, the curly braces also help with readability with single code lines.*

With the `if` statement, `else if` and `else` are optional. If you just need to invoke a code block based on a condition and don't invoke a code block if this condition is not met, you can use the `if` without `else`.

Pattern Matching with the is Operator

One of the C# features is *pattern matching*, which you can use with the `if` statement and the `is` operator. The earlier section "Nullable Reference Types" included an example that used an `if` statement and the pattern `is not null`.

The following code snippet compares the argument received that is of type `object` with `null`, using a *const pattern* to compare the argument with `null` and throw the `ArgumentNullException`. With the expression used in

`else if`, the *type pattern* is used to check whether the variable `o` is of type `Book`. If this is the case, the variable `o` is assigned to the variable `b`. Because variable `b` is of type `Book`, with `b` the `Title` property that is specified by the `Book` type can be accessed (code file `ProgramFlow/Program.cs`):

```
void PatternMatching(object o)
{
  if (o is null) throw new ArgumentNullException(nameof(o));
  else if (o is Book b)
  {
    Console.WriteLine($"received a book: {b.Title}");
  }
}
```

> **NOTE** *In this example, for throwing the* `ArgumentNullException`*, the* `nameof` *expression is used. The* `nameof` *expression is resolved from the compiler to take the name of the argument—for example, the variable* `o`*—and pass it as a string.* `throw new ArgumentNullException(nameof(o));` *resolves to the same code as* `throw new ArgumentNullException("o");`*. However, if the variable* `o` *is renamed to a different value, refactoring features can automatically rename the variable specified with the* `nameof` *expression. If the parameter of* `nameof` *is not changed when the variable is renamed, a compiler error will be the result. Without the* `nameof` *expression, the variable and the string can easily get out of sync.*

A few more samples for const and type patterns are shown in the following code snippet:

```
if (o is 42) // const pattern
if (o is "42") // const pattern
if (o is int i) // type pattern
```

> **NOTE** *You can use pattern matching with the* `is` *operator, the* `switch` *statement, and the* `switch` *expression. You can use different categories of pattern matching. This chapter only covers const, type, relational patterns, and pattern combinators. More patterns, such as property patterns, patterns with tuples, and recursive patterns, are covered in Chapter 3.*

The switch Statement

The `switch`/`case` statement is good for selecting one branch of execution from a set of mutually exclusive ones. It takes the form of a `switch` argument followed by a series of `case` clauses. When the expression in the `switch` argument evaluates to one of the values specified by a `case` clause, the code immediately following the `case` clause executes. This is one example for which you don't need to use curly braces to join statements into blocks; instead, you mark the end of the code for each case using the `break` statement. You can also include a `default` case in the `switch` statement, which executes if the expression doesn't evaluate to any of the other cases. The following `switch` statement tests the value of the `x` variable (code file `SwitchStatement/Program.cs`):

```
void SwitchSample(int x)
{
  switch (x)
  {
    case 1:
      Console.WriteLine("integerA = 1");
```

```
        break;
      case 2:
        Console.WriteLine("integerA = 2");
        break;
      case 3:
        Console.WriteLine("integerA = 3");
        break;
      default:
        Console.WriteLine("integerA is not 1, 2, or 3");
        break;
    }
}
```

Note that the case values must be constant expressions; variables are not permitted.

With the `switch` statement, you cannot remove the `break` from the different cases. Contrary to the C++ and Java programming languages, with C# a*utomatic fall-through* from one `case` implementation to continue with another `case` is not done. Instead of an automatic fall-through, you can use the `goto` keyword for an explicit fall-through and select another case. Here's an example:

```
goto case 3;
```

If the implementation is completely the same with multiple cases, you can specify multiple cases before specifying an implementation:

```
switch(country)
{
  case "au":
  case "uk":
  case "us":
    language = "English";
    break;
  case "at":
  case "de":
    language = "German";
    break;
}
```

Pattern Matching with the switch Statement

Pattern matching can also be used with the `switch` statement. The following code snippet shows different `case` options with const and type, and relational patterns. The method `SwitchWithPatternMatching` receives a parameter of type `object`. `case null` is a const pattern that compares `o` for `null`. The next three cases specify a type pattern. `case int i` uses a type pattern that creates the variable `i` if `o` is an `int`, but only in combination with the `when` clause. The `when` clause uses a relational pattern to check if it is larger than `42`. The next case matches every remaining `int` type. Here, no variable is specified where object `o` should be assigned. Specifying a variable is not necessary if you don't need this variable and just need to know it's of this type. With the match for a `Book` type, the variable `b` is used. Declaring a variable here, this variable is of type `Book` (code file `SwitchStatement/Program.cs`):

```
void SwitchWithPatternMatching(object o)
{
  switch (o)
  {
    case null:
      Console.WriteLine("const pattern with null");
```

```
        break;
      case int i when i > 42
        Console.WriteLine("type pattern with when and a relational pattern");
      case int:
        Console.WriteLine("type pattern with an int");
        break;
      case Book b:
        Console.WriteLine($"type pattern with a Book {b.Title}");
        break;
      default:
        break;
    }
}
```

The switch Expression

The next example shows a `switch` based on an `enum` type. The `enum` type is based on an integer but gives names to the different values. The type `TrafficLight` defines the different values for the colors of a traffic light (code file `SwitchExpression/Program.cs`):

```
enum TrafficLight
{
  Red,
  Amber,
  Green
}
```

> **NOTE** *Chapter 3 goes into more detail about the* `enum` *type so you can see the effect of changing the base type and assigning different values.*

With the `switch` statement so far, you've only seen invoking some actions in every case. When you use the `return` statement to return from a method, you can also directly return a value from the `case` without continuing with the following cases. The method `NextLightClassic` receives a `TrafficLight` with its parameter and returns a `TrafficLight`. If the passed traffic light has the value `TrafficLight.Green`, the method returns `TrafficLight.Amber`. When the current light value is `TrafficLight.Amber`, `TrafficLight.Red` is returned:

```
TrafficLight NextLightClassic(TrafficLight light)
{
  switch (light)
  {
    case TrafficLight.Green:
      return TrafficLight.Amber;
    case TrafficLight.Amber:
      return TrafficLight.Red;
    case TrafficLight.Red:
      return TrafficLight.Green;
    default:
       throw new InvalidOperationException();
  }
}
```

In such a scenario, if you need to return a value based on different options, you can use the *switch expression* that is new as of C# 8. The method `NextLight` receives and returns a `TrafficLight` value similar to the previously shown method. The implementation is now done with an expression bodied member because the implementation is done in a single statement. Curly braces and the `return` statement are unnecessary in this case. When you use a `switch` expression instead of the `switch` statement, the variable and `switch` keyword are reversed. With the `switch` statement, the value on the switch follows in braces after the `switch` keyword. With the `switch` expression, the variable is followed by the `switch` keyword. A block with curly braces defines the different cases. Instead of using the `case` keyword, the `=>` token is used to define what's returned. The functionality is the same as before, but you need fewer lines of code:

```
TrafficLight NextLight(TrafficLight light) =>
  light switch
  {
    TrafficLight.Green => TrafficLight.Amber,
    TrafficLight.Amber => TrafficLight.Red,
    TrafficLight.Red => TrafficLight.Green,
    _ => throw new InvalidOperationException()
  };
```

If the `enum` type `TrafficLight` is imported with the `using static` directive, you can simplify the implementation even more by just using the `enum` value definitions without the type name:

```
using static TrafficLight;

TrafficLight NextLight(TrafficLight light) =>
  light switch
  {
    Green => Amber,
    Amber => Red,
    Red => Green,
    _ => throw new InvalidOperationException()
  };
```

> **NOTE** *In the United States, the switch on the traffic light is simple compared to many other countries. In many countries, the light switches from red back to amber. Here you could use multiple amber states such as* `AmberAfterGreen` *and* `AmberAfterRed`*. But there are other options that require the property pattern or pattern matching based on tuples. This is covered in Chapter 3.*

With the next example, a pattern combinator is used to combine multiple patterns. First, input is retrieved from the console. If string `one` or `two` is entered, the same match applies, using the `or` combinator pattern (code file `SwitchExpression/Program.cs`):

```
string? input = Console.ReadLine();

string result = input switch
{
  "one" => "the input has the value one",
  "two" or "three" => "the input has the value two or three",
  _ => "any other value"
};
```

With pattern combinators, you can combine patterns using the `and`, `or`, and `not` keywords.

The for Loop

C# provides four different loops (`for`, `while`, `do-while`, and `foreach`) that enable you to execute a block of code repeatedly until a certain condition is met. With the `for` keyword, you iterate through a loop whereby you test whether a particular condition holds true before you perform another iteration:

```
for (int i = 0; i < 100; i++)
{
  Console.WriteLine(i);
}
```

The first expression of the `for` statement is the *initializer*. It is evaluated before the first loop is executed. Usually you use this to initialize a local variable as a loop counter.

The second expression is the *condition*. This is checked before every iteration of the `for` block. If this expression evaluates to `true`, the block is executed. If it evaluates to `false`, the `for` statement ends, and the program continues with the next statement after the closing curly brace of the `for` body.

After the body is executed, the third expression, the *iterator*, is evaluated. Usually, you increment the loop counter. With `i++`, a value of 1 is added to the variable `i`. After the third expression, the condition expression is evaluated again to check whether another iteration with the `for` block should be done.

The `for` loop is a so-called pretest loop because the loop condition is evaluated before the loop statements are executed; therefore, the contents of the loop won't be executed at all if the loop condition is `false`.

It's not unusual to nest `for` loops so that an inner loop executes once completely for each iteration of an outer loop. This approach is typically employed to loop through every element in a rectangular multidimensional array. The outermost loop loops through every row, and the inner loop loops through every column in a particular row. The following code displays rows of numbers. It also uses another `Console` method, `Console.Write`, which does the same thing as `Console.WriteLine` but doesn't send a carriage return to the output (code file `ForLoop/Program.cs`):

```
// This loop iterates through rows
for (int i = 0; i < 100; i += 10)
{
  // This loop iterates through columns
  for (int j = i; j < i + 10; j++)
  {
    Console.Write($" {j}");
  }
  Console.WriteLine();
}
```

This sample results in this output:

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
80 81 82 83 84 85 86 87 88 89
90 91 92 93 94 95 96 97 98 99
```

> **NOTE** *It is technically possible to evaluate something other than a counter variable in a* `for` *loop's test condition, but it is certainly not typical. It is also possible to omit one (or even all) of the expressions in the* `for` *loop. In such situations, however, you should consider using the* `while` *loop.*

The while Loop

Like the `for` loop, `while` is a pretest loop. The syntax is similar, but `while` loops take only one expression:

```
while(condition)
  statement(s);
```

Unlike the `for` loop, the `while` loop is most often used to repeat a statement or a block of statements for a number of times that is not known before the loop begins. Usually, a statement inside the `while` loop's body sets a Boolean flag to `false` on a certain iteration, triggering the end of the loop, as in the following example:

```
bool condition = false;
while (!condition)
{
  // This loop spins until the condition is true.
  DoSomeWork();
  condition = CheckCondition(); // assume CheckCondition() returns a bool
}
```

The do-while Loop

The `do-while` loop is the post-test version of the `while` loop. This means that the loop's test condition is evaluated after the body of the loop has been executed. Consequently, `do-while` loops are useful for situations in which a block of statements must be executed at least one time, as in this example:

```
bool condition;
do
{
  // This loop will at least execute once, even if the condition is false.
  MustBeCalledAtLeastOnce();
  condition = CheckCondition();
} while (condition);
```

The foreach Loop

The `foreach` loop enables you to iterate through each item in a collection. For now, don't worry about exactly what a collection is (it is explained fully in Chapter 6, "Arrays"); just understand that it is an object that represents a list of objects. Technically, for an object to count as a collection, it must support an interface called `IEnumerable`. Examples of collections include C# arrays, the collection classes in the `System.Collections` namespaces, and user-defined collection classes. You can get an idea of the syntax of `foreach` from the following code, if you assume that `arrayOfInts` is (unsurprisingly) an array of `int`s:

```
foreach (int temp in arrayOfInts)
{
  Console.WriteLine(temp);
}
```

Here, `foreach` steps through the array one element at a time. With each element, it places the value of the element in the `int` variable called `temp` and then performs an iteration of the loop.

Here is another situation where you can use type inference. The `foreach` loop would become the following:

```
foreach (var temp in arrayOfInts)
{
  // ...
}
```

`int` would infer from `temp` because that is what the collection item type is.

An important point to note with `foreach` is that you can't change the value of the item in the collection (`temp` in the preceding code), so code such as the following will not compile:

```
foreach (int temp in arrayOfInts)
{
  temp++;
  Console.WriteLine(temp);
}
```

If you need to iterate through the items in a collection and change their values, you must use a `for` loop instead.

Exiting Loops

Within a loop, you can stop the iterations with the `break` statement or end the current iteration and continue with the next iteration with the `continue` statement. With the `return` statement, you can exit the current method and thus also exit a loop.

ORGANIZATION WITH NAMESPACES

With small sample applications, you don't need to specify a namespace. When you create libraries where classes are used in applications, to avoid ambiguities, you must specify namespaces. The `Console` class used earlier is defined in the `System` namespace. To use the class `Console`, you either have to prefix it with the namespace or import the namespace from this class.

Namespaces can be defined in a hierarchical way. For example, the `ServiceCollection` class is specified in the namespace `Microsoft.Extensions.DependencyInjection`. To define the class `Sample` in the namespace `Wrox.ProCSharp.CoreCSharp`, you can specify this namespace hierarchy with the `namespace` keyword:

```
namespace Wrox
{
  namespace ProCSharp
  {
    namespace CoreCSharp
    {
      public class Sample
      {
      }
    }
  }
}
```

You can also use the dotted notation to specify the namespace:

```
namespace Wrox.ProCSharp.CoreCSharp
{
```

```
    public class Sample
    {
    }
}
```

A namespace is a logical construct and completely independent of physical files or components. One assembly can contain multiple namespaces, and a single namespace can be spread across multiple assemblies. It's a logical construct to group different types together.

Each namespace name is composed of the names of the namespaces it resides within, separated with periods, starting with the outermost namespace and ending with its own short name. Therefore, the full name for the `ProCSharp` namespace is `Wrox.ProCSharp`, and the full name of the `Sample` class is `Wrox.ProCSharp.CoreCSharp.Sample`.

The using Directive

Obviously, namespaces can grow rather long and tiresome to type, and the capability to indicate a particular class with such specificity may not always be necessary. Fortunately, as noted earlier in this chapter, C# allows you to abbreviate a class's full name. To do this, list the class's namespace at the top of the file, prefixed with the `using` keyword. Throughout the rest of the file, you can refer to the types in the namespace by their type names.

If two namespaces referenced by `using` declarations contain a type of the same name, you need to use the full (or at least a longer) form of the name to ensure that the compiler knows which type to access. For example, suppose classes called `Test` exist in both the `ProCSharp.CoreCSharp` and `ProCSharp.OOP` namespaces. If you then create a class called `Test` and both namespaces are imported, the compiler reacts with an ambiguity compilation error. In this case, you need to specify the namespace name for the type.

Namespace Aliases

Instead of specifying the complete namespace name for the class to resolve ambiguity issues, you can specify an alias with the `using` directive, as shown with different `Timer` classes from two namespaces:

```
using TimersTimer = System.Timers.Timer;
using Webtimer = System.Web.UI.Timer;
```

WORKING WITH STRINGS

The code in this chapter has already used the `string` type several times. `string` is an important reference type that offers many features. Although it's a reference type, it's immutable—it can't be changed. All the methods this type offers don't change the content of the string but instead return a new string. For example, to concatenate strings, the `+` operator is overloaded. The expression `s1 + " " + s2` first creates a new string combining `s1` and the string containing the space character. Another new string is created by combining the result string with `s2` to create another new string. Finally, the result string is referenced from the variable `s3`:

```
string s1 = "Hello";
string s2 = "World";
string s3 = s1 + "  " + s2;
```

With many strings created, you need to be aware that the objects that are no longer necessary need to be cleaned up by the garbage collector. The garbage collector frees up memory in the managed heap from objects that are no longer needed. This doesn't happen when the reference is not used anymore; it's based on certain memory limits. Read Chapter 13 for more information on the garbage collector. It's best to avoid object allocation, which can be done when dynamically working with strings by using the `StringBuilder` class.

Using the StringBuilder

The `StringBuilder` allows a program to dynamically work with strings using `Append`, `Insert`, `Remove`, and `Replace` methods without creating new objects. Instead, the `StringBuilder` uses a memory buffer and modifies this buffer as the need arises. When you're creating a `StringBuilder`, the default capacity is 16 characters. If strings are appended as shown in the following code snippet and more memory is needed, the capacity is doubled to 32 characters (code file `StringSample/Program.cs`):

```
void UsingStringBuilder()
{
  StringBuilder sb = new("the quick");
  sb.Append(' ');
  sb.Append("brown fox jumped over ");
  sb.Append("the lazy dogs 1234567890 times");
  string s = sb.ToString();
  Console.WriteLine(s);
}
```

If the capacity is too small, the buffer size always doubles—for example, from 16 to 32 to 64 to 128 characters. The length of the string can be accessed with the `Length` property. The capacity of the `StringBuilder` is returned from the `Capacity` property. After creating the necessary string, you can use the `ToString` method, which allocates a new string containing the content of the `StringBuilder`.

String Interpolation

Code snippets in this chapter have already included strings with the `$` prefix. This prefix allows evaluating expressions within the string and is known as *string interpolation*. For example, with string `s2`, the content of string `s1` is embedded within `s2` to have the final result of `Hello, World!`:

```
string s1 = "World";
string s2 = $"Hello, {s1}!";
```

You can write code expressions within the curly braces to get the expression evaluated and the result added into the string. In the following code snippet, a string is specified with three placeholders where the value of `x`, the value of `y`, and the result of the addition of `x` and `y` are put into the string:

```
int x = 3, y = 4;
string s3 = $"The result of {x} and {y} is {x + y}";
Console.WriteLine(s3);
```

The resulting string is `The result of 3 and 4 is 7`.

The compiler translates the interpolated string to invoke the `Format` method of the string, passes a string with numbered placeholders, and passes additional arguments following the string. The result of the additional arguments is from the implementation of the `Format` method passed to the placeholders based on the numbers. The first argument following the string is passed to the 0 placeholder, the second argument to the 1 placeholder, and so on:

```
string s3 = string.Format("The result of {0} and {1} is {2}", x, y, x + y);
```

> **NOTE** *To escape curly braces in an interpolated string, you can use double curly braces:* `{{}}`.

FormattableString

What the interpolated string gets translated to can easily be seen by assigning a string to a `FormattableString`. The interpolated string can be directly assigned to this type because it's a better match than the normal string. This type defines the `Format` property that returns the resulting format string, an `ArgumentCount` property, and the method `GetArgument` that returns the argument values (code file `StringSample/Program.cs`):

```
void UsingFormattableString()
{
  int x = 3, y = 4;
  FormattableString s = $"The result of {x} + {y} is {x + y}";
  Console.WriteLine($"format: {s.Format}");
  for (int i = 0; i < s.ArgumentCount; i++)
  {
    Console.WriteLine($"argument: {i}:{s.GetArgument(i)}");
  }
  Console.WriteLine();
}
```

Running this code snippet results in this output:

```
format: The result of {0} + {1} is {2}
argument 0: 3
argument 1: 4
argument 2: 7
```

> **NOTE** *In Chapter 22, "Localization," you can read about using string interpolation with different cultures. By default, string interpolation makes use of the current culture.*

String Formats

With an interpolated string, you can add a string format to the expression. .NET defines default formats for numbers, dates, and time based on the computer's locale. The following code snippet shows a date, an `int` value, and a `double` with different format representations. `D` is used to display the date in the long date format, `d` in the short date format. The number is shown with integral and decimal digits (`n`), using an exponential notation (`e`), a conversion to hexadecimal (`x`), and a currency (`c`). With the double value, the first result is shown rounded after the decimal point to three digits (`###.###`); with the second version, the three digits before the decimal point are shown as well (`000.000`):

```
void UseStringFormat()
{
  DateTime day = new(2025, 2, 14);
  Console.WriteLine($"{day:D}");
  Console.WriteLine($"{day:d}");

  int i = 2477;
  Console.WriteLine($"{i:n} {i:e} {i:x} {i:c}");

  double d = 3.1415;
  Console.WriteLine($"{d:###.###}");
  Console.WriteLine($"{d:000.000}");
  Console.WriteLine();
}
```

When you run the application, this is shown:

```
Friday, February 14, 2025
2/14/2025
2,477.00 2.477000e+003 9ad $2,477.00
3.142
```

> **NOTE** *See the Microsoft documentation for all the different format strings for numbers at* `https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-numeric-format-strings` *and for date/time at* `https://docs.microsoft.com/en-us/dotnet/standard/base-types/standard-date-and-time-format-strings`. *To define custom formats with a custom type, see the sample in Chapter 9, "Language Integrated Query."*

Verbatim Strings

Code snippets in the section "The Character Type" earlier in this chapter included special characters such as `\t` for a tab or `\r\n` for carriage return newline. You can use these characters in a complete string to get the specific meaning. If you need a backslash in the output of the string, you can escape this with a double backslash `\\`. This can be annoying if backslashes are needed multiple times because they can make the code unreadable. For such scenarios, such as when using regular expressions, you can use verbatim strings. A verbatim string is prefixed with the `@` character:

```
string s = @"a tab: \t, a carriage return: \r, a newline: \n";
Console.WriteLine(s);
```

Running the preceding code results in this output:

```
a tab: \t, a carriage return: \r, a newline: \n
```

Ranges with Strings

The `String` type offers a `Substring` method to retrieve a part of a string. Instead of using the `Substring` method, as of C# 8 you can use the *hat* and the *range* operators. The range operator uses the `..` notation to specify a range. With the string, you can use the indexer to access one character or use it with the range operator to access a substring. The numbers left and right of the `..` operator specify the range. The left number specifies the 0-indexed first value from the string, which is included from the string up to the 0-indexed last value that is excluded. The range `0..3` would span the string `The`. To start from the first character in the string, the `0` can be omitted as shown with the following code snippet. The range `4..9` starts with the fifth character and goes up to the eighth character. To count from the end, you can use the hat operator `^` (code file `StringSample/Program.cs`):

```
void RangesWithStrings()
{
  string s = "The quick brown fox jumped over the lazy dogs down " +
    "1234567890 times";
  string the = s[..3];
  string quick = s[4..9];
  string times = s[^5..^0];
  Console.WriteLine(the);
  Console.WriteLine(quick);
  Console.WriteLine(times);
  Console.WriteLine();
}
```

> **NOTE** *For more information about the indices, ranges, and the hat operator, read Chapter 6, "Arrays."*

COMMENTS

The next topic—adding comments to your code—looks simple on the surface, but it can be complex. Comments can be beneficial to other developers who may look at your code. Also, as you will see, you can use comments to generate documentation for your code that other developers can use.

Internal Comments Within the Source Files

C# uses the traditional C-type single-line (`//..`) and multiline (`/* .. */`) comments:

```
// This is a single-line comment
/* This comment
spans multiple lines. */
```

Everything in a single-line comment, from the `//` to the end of the line, is ignored by the compiler, and everything from an opening `/*` to the next `*/` in a multiline comment combination is ignored. It is possible to put multiline comments within a line of code:

```
Console.WriteLine(/* Here's a comment! */ "This will compile.");
```

Inline comments can be useful when debugging if, for example, you temporarily want to try running the code with a different value somewhere, as in the following code snippet. However, inline comments can make code hard to read, so use them with care.

```
DoSomething(Width, /*Height*/ 100);
```

XML Documentation

In addition to the C-type comments illustrated in the preceding section, C# has a very neat feature: the capability to produce documentation in XML format automatically from special comments. These comments are single-line comments, but they begin with three slashes (`///`) instead of two. Within these comments, you can place XML tags containing documentation of the types and type members in your code.

The tags in the following table are recognized by the compiler:

TAG	DESCRIPTION
`<c>`	Marks up text within a line as code—for example, `<c>int i = 10;</c>`.
`<code>`	Marks multiple lines as code.
`<example>`	Marks up a code example.
`<exception>`	Documents an exception class. (Syntax is verified by the compiler.)
`<include>`	Includes comments from another documentation file. (Syntax is verified by the compiler.)
`<list>`	Inserts a list into the documentation.
`<para>`	Gives structure to text.

TAG	DESCRIPTION
<param>	Marks up a method parameter. (Syntax is verified by the compiler.)
<paramref>	Indicates that a word is a method parameter. (Syntax is verified by the compiler.)
<permission>	Documents access to a member. (Syntax is verified by the compiler.)
<remarks>	Adds a description for a member.
<returns>	Documents the return value for a method.
<see>	Provides a cross-reference to another parameter. (Syntax is verified by the compiler.)
<seealso>	Provides a "see also" section in a description. (Syntax is verified by the compiler.)
<summary>	Provides a short summary of a type or member.
<typeparam>	Describes a type parameter in the comment of a generic type.
<typeparamref>	Provides the name of the type parameter.
<value>	Describes a property.

The following code snippet shows the `Calculator` class with documentation specified for the class, and documentation for the `Add` method (code file `Math/Calculator.cs`):

```
namespace ProCSharp.MathLib
{
  ///<summary>
  /// ProCsharp.MathLib.Calculator class.
  /// Provides a method to add two doubles.
  ///</summary>
  public static class Calculator
  {
    ///<summary>
    /// The Add method allows us to add two doubles.
    ///</summary>
    ///<returns>Result of the addition (double)</returns>
    ///<param name="x">First number to add</param>
    ///<param name="y">Second number to add</param>
    public static double Add(double x, double y) => x + y;
  }
}
```

To generate the XML documentation, you can add the `GenerateDocumentationFile` to the project file (project configuration file `Math/Math.csproj`):

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>exe</OutputType>
    <TargetFramework>net5.0</TargetFramework>
    <Nullable>enable</Nullable>
    <GenerateDocumentationFile>true</GenerateDocumentationFile>
  </PropertyGroup>

</Project>
```

With this setting, the documentation file is created in the same directory where the program binary will show up as you compile the application. You can also specify the `DocumentationFile` element to define a name that's different from the project file, and you can also specify an absolute directory where the documentation should be generated.

Using tools like Visual Studio, IntelliSense will show tooltips with the information from the documentation as the classes and members are used.

C# PREPROCESSOR DIRECTIVES

Besides the C# keywords, most of which you have now encountered, C# includes a number of commands that are known as *preprocessor directives*. These commands are never actually translated to any commands in your executable code, but they affect aspects of the compilation process. For example, you can use preprocessor directives to prevent the compiler from compiling certain portions of your code. You might do this if you target different frameworks and deal with the differences. In another scenario, you might want to turn nullable reference types on or off because changing existing codebases cannot be fixed in the short term.

The preprocessor directives are all distinguished by beginning with the `#` symbol.

The following sections briefly cover the purposes of the preprocessor directives.

#define and #undef

`#define` is used like this:

```
#define DEBUG
```

This tells the compiler that a symbol with the given name (in this case `DEBUG`) exists. It is a little bit like declaring a variable, except that this variable doesn't really have a value—it just exists. Also, this symbol isn't part of your actual code; it exists only for the benefit of the compiler, whereas the compiler is compiling the code and has no meaning within the C# code itself.

`#undef` does the opposite and removes the definition of a symbol:

```
#undef DEBUG
```

If the symbol doesn't exist in the first place, then `#undef` has no effect. Similarly, `#define` has no effect if a symbol already exists.

You need to place any `#define` and `#undef` directives at the beginning of the C# source file, before any code that declares any objects to be compiled.

`#define` isn't of much use on its own, but when combined with other preprocessor directives, especially `#if`, it becomes powerful.

By default, with a Debug build, the `DEBUG` symbol is defined, and with the Release code, the `RELEASE` symbol is defined. To define different code paths on debug and release builds, you don't need to define these symbols; all you have to do is to use the preprocessor directives shown in the next section to define the code paths the compiler should take.

> **NOTE** *Preprocessor directives are not terminated by semicolons, and they normally constitute the only command on a line. If the compiler sees a preprocessor directive, it assumes that the next command is on the next line.*

#if, #elif, #else, and #endif

These directives inform the compiler whether to compile a block of code. Consider this method:

```
int DoSomeWork(double x)
{
  // do something
  #if DEBUG
  Console.WriteLine($"x is {x}");
  #endif
}
```

This code compiles as normal except for the `Console.WriteLine` method call contained inside the `#if` clause. This line is executed only if the symbol `DEBUG` has been defined. As previously mentioned, it's defined with a Debug build—or you defined it with a previous `#define` directive. When the compiler finds the `#if` directive, it checks to see whether the symbol concerned exists and compiles the code inside the `#if` clause only if the symbol does exist. Otherwise, the compiler simply ignores all the code until it reaches the matching `#endif` directive. Typical practice is to define the symbol `DEBUG` while you are debugging and have various bits of debugging-related code inside `#if` clauses. Then, when you are close to shipping, you simply comment out the `#define` directive, and all the debugging code miraculously disappears, the size of the executable file gets smaller, and your end users don't get confused by seeing debugging information. (Obviously, you would do more testing to ensure that your code still works without `DEBUG` defined.) This technique is common in C and C++ programming and is known as *conditional compilation*.

The `#elif` (`=else if`) and `#else` directives can be used in `#if` blocks and have intuitively obvious meanings. It is also possible to nest `#if` blocks:

```
#define ENTERPRISE
#define W10
// further on in the file
#if ENTERPRISE
// do something
#if W10
// some code that is only relevant to enterprise
// edition running on W10
#endif
#elif PROFESSIONAL
// do something else
#else
// code for the leaner version
#endif
```

`#if` and `#elif` support a limited range of logical operators, too, using the operators `!`, `==`, `!=`, `&&`, and `||`. A symbol is considered to be `true` if it exists and `false` if it doesn't. Here's an example:

```
#if W10 && !ENTERPRISE // if W10 is defined but ENTERPRISE isn't
```

#warning and #error

Two other useful preprocessor directives, `#warning` and `#error`, cause a warning or an error, respectively, to be raised when the compiler encounters them. If the compiler sees a `#warning` directive, it displays whatever text appears after the `#warning` to the user, after which compilation continues. If it encounters an `#error` directive, it displays the subsequent text to the user as if it is a compilation error message and then immediately abandons the compilation, so no IL code is generated.

You can use these directives as checks that you haven't done anything silly with your `#define` statements; you can also use the `#warning` statements to remind yourself to do something:

```
#if DEBUG && RELEASE
#error "You've defined DEBUG and RELEASE simultaneously!"
#endif

#warning "Don't forget to remove this line before the boss tests the code!"
Console.WriteLine("*I love this job.*");
```

#region and #endregion

The `#region` and `#endregion` directives are used to indicate that a certain block of code is to be treated as a single block with a given name, like this:

```
#region Member Field Declarations
int x;
double d;
decimal balance;
#endregion
```

The region directives are ignored by the compiler and used by tools such as the Visual Studio code editor. The editor allows you to collapse region sections, so only the text associated with the region shows. This makes it easier to scroll through the source code. However, you should prefer to write shorter code files instead.

#line

You can use the `#line` directive to alter the filename and line number information that is output by the compiler in warnings and error messages. You probably won't want to use this directive often. It's most useful when you are coding in conjunction with another package that alters the code you are typing before sending it to the compiler. In this situation, line numbers, or perhaps the filenames reported by the compiler, don't match up to the line numbers in the files or the filenames you are editing. The `#line` directive can be used to restore the match. You can also use the syntax `#line default` to restore the line to the default line numbering:

```
#line 164 "Core.cs" // We happen to know this is line 164 in the file
// Core.cs, before the intermediate
// package mangles it.
// later on
#line default // restores default line numbering
```

#pragma

The `#pragma` directive can either suppress or restore specific compiler warnings. Unlike command-line options, the `#pragma` directive can be implemented on the class or method level, enabling fine-grained control over what warnings are suppressed and when. The following example disables the "field not used" warning and then restores it after the `MyClass` class compiles:

```
#pragma warning disable 169
public class MyClass
{
  int neverUsedField;
}
#pragma warning restore 169
```

#nullable

With the `#nullable` directive, you can turn on or off nullable reference types within a code file. `#nullable enable` turns nullable reference types on, no matter what the setting in the project file. `#nullable disable` turns it off. `#nullable restore` switches the settings back to the settings of the project file.

How do you use this? If nullable reference types are enabled with the project file, you can temporarily turn them off in code sections where you have issues with this compiler behavior and restore it to the project file settings after the code with nullability issues.

C# PROGRAMMING GUIDELINES

This final section of the chapter supplies the guidelines you need to bear in mind when writing C# programs. These are guidelines that most C# developers use. When you use these guidelines, other developers will feel comfortable working with your code.

Rules for Identifiers

This section examines the rules governing what names you can use for variables, classes, methods, and so on. Note that the rules presented in this section are not merely guidelines: they are enforced by the C# compiler.

Identifiers are the names you give to variables, user-defined types such as classes and structs, and members of these types. Identifiers are case sensitive, so, for example, variables named `interestRate` and `InterestRate` would be recognized as different variables. The following are a few rules determining what identifiers you can use in C#:

- They must begin with a letter or underscore, although they can contain numeric characters.
- You can't use C# keywords as identifiers.

See the list of C# reserved keywords at `https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/`.

If you need to use one of these words as an identifier (for example, if you are accessing a class written in a different language), you can prefix the identifier with the `@` symbol to indicate to the compiler that what follows should be treated as an identifier, not as a C# keyword (so `abstract` is not a valid identifier, but `@abstract` is).

Finally, identifiers can also contain Unicode characters, specified using the syntax `\uXXXX`, where `XXXX` is the four-digit hex code for the Unicode character. The following are some examples of valid identifiers:

- `Name`
- `Überfluß`
- `_Identifier`
- `\u005fIdentifier`

The last two items in this list are identical and interchangeable (because `005f` is the Unicode code for the underscore character), so, obviously, both these identifiers couldn't be declared in the same scope.

Usage Conventions

In any development language, certain traditional programming styles usually arise. The styles are not part of the language itself but rather are conventions—for example, how variables are named or how certain classes, methods, or functions are used. If most developers using that language follow the same conventions, it's easier

for different developers to understand each other's code—which in turn generally helps program maintainability. Conventions do, however, depend on the language and the environment. For example, C++ developers programming on the Windows platform have traditionally used the prefixes `psz` or `lpsz` to indicate strings—`char *pszResult; char *lpszMessage;`—but on Unix machines it's more common not to use any such prefixes: `char *Result; char *Message;`.

> **NOTE** *The convention by which variable names are prefixed with letters that represent the data type is known as Hungarian notation. It means that other developers reading the code can immediately tell from the variable name what data type the variable represents. Hungarian notation is widely regarded as redundant in these days of smart editors and IntelliSense.*

Whereas many languages' usage conventions simply evolved as the language was used, for C# and the whole of the .NET Framework, Microsoft has written comprehensive usage guidelines that are detailed in the .NET/C# documentation. This means that, right from the start, .NET programs have a high degree of interoperability in terms of developers being able to understand code. The guidelines have also been developed with the benefit of some 20 years' hindsight in object-oriented programming. Judging by the relevant newsgroups, the guidelines have been carefully thought out and are well received in the developer community. Hence, the guidelines are well worth following.

Note, however, that the guidelines are not the same as language specifications. You should try to follow the guidelines when you can. Nevertheless, you won't run into problems if you have a good reason for not doing so—for example, you won't get a compilation error because you don't follow these guidelines. The general rule is that if you don't follow the usage guidelines, you must have a convincing reason. When you depart from the guidelines, you should be making a conscious decision rather than simply not bothering. Also, if you compare the guidelines with the samples in the remainder of this book, you'll notice that in numerous examples I have chosen not to follow the conventions. That's usually because the conventions are designed for much larger programs than the samples; although the guidelines are great if you are writing a complete software package, they're not really suitable for small 20-line stand-alone programs. In many cases, following the conventions would have made the samples harder, rather than easier, to follow.

The full guidelines for good programming style are quite extensive. This section is confined to describing some of the more important guidelines, as well as those most likely to surprise you. To be absolutely certain that your code follows the usage guidelines completely, you need to refer to the Microsoft documentation.

Naming Conventions

One important aspect of making your programs understandable is how you choose to name your items—and that includes naming variables, methods, classes, enumerations, and namespaces.

It is intuitively obvious that your names should reflect the purpose of the item and should not clash with other names. The general philosophy in the .NET Framework is also that the name of a variable should reflect the purpose of that variable instance and not the data type. For example, `height` is a good name for a variable, whereas `integerValue` isn't. However, you are likely to find that principle is an ideal that is hard to achieve. Particularly when you are dealing with controls, in most cases you'll probably be happier sticking with variable names such as `confirmationDialog` and `chooseEmployeeListBox`, which do indicate the data type in the name.

The following sections look at some of the things you need to think about when choosing names.

Casing of Names

In many cases, you should use *Pascal casing* for names. With Pascal casing, the first letter of each word in a name is capitalized: `EmployeeSalary`, `ConfirmationDialog`, `PlainTextEncoding`. Notice that nearly all the names of

namespaces, classes, and members in the base classes follow Pascal casing. In particular, the convention of joining words using the underscore character is discouraged. Therefore, try not to use names such as `employee_salary`. It has also been common in other languages to use all capitals for names of constants. This is not advised in C# because such names are harder to read—the convention is to use Pascal casing throughout:

```
const int MaximumLength;
```

The only other casing convention that you are advised to use is *camel casing*. Camel casing is similar to Pascal casing, except that the first letter of the first word in the name is not capitalized: `employeeSalary`, `confirmationDialog`, `plainTextEncoding`. The following are three situations in which you are advised to use camel casing:

- For names of all private member fields in types.
- For names of all parameters passed to methods.
- To distinguish items that would otherwise have the same name. A common example is when a property wraps around a field:

```
private string employeeName;
public string EmployeeName
{
  get
  {
    return employeeName;
  }
}
```

> **NOTE** *Since .NET Core, the .NET team has been prefixing names of private member fields with an underscore. This is also used as a convention with this book.*

If you are wrapping a property around a field, you should always use camel casing for the private member and Pascal casing for the public or protected member so that other classes that use your code see only names in Pascal case (except for parameter names).

You should also be wary about case sensitivity. C# is case sensitive, so it is syntactically correct for names in C# to differ only by the case, as in the previous examples. However, bear in mind that your assemblies might at some point be called from Visual Basic applications—and *Visual Basic is not case sensitive*. Hence, if you do use names that differ only by case, it is important to do so only in situations in which both names will never be seen outside your assembly. (The previous example qualifies as okay because camel case is used with the name that is attached to a `private` variable.) Otherwise, you may prevent other code written in Visual Basic from being able to use your assembly correctly.

Name Styles

Be consistent about your style of names. For example, if one of the methods in a class is called `ShowConfirmationDialog`, then you should not give another method a name such as `ShowDialogWarning` or `WarningDialogShow`. The other method should be called `ShowWarningDialog`.

Namespace Names

It is particularly important to choose namespace names carefully to avoid the risk of ending up with the same name for one of your namespaces as someone else uses. Remember, namespace names are the *only* way that .NET distinguishes names of objects in shared assemblies. Therefore, if you use the same namespace name for your

software package as another package and both packages are used by the same program, problems will occur. Because of this, it's almost always a good idea to create a top-level namespace with the name of your company and then nest successive namespaces that narrow down the technology, group, or department you are working in or the name of the package for which your classes are intended. Microsoft recommends namespace names that begin with `<CompanyName>.<TechnologyName>`.

Names and Keywords

It is important that the names do not clash with any keywords. In fact, if you attempt to name an item in your code with a word that happens to be a C# keyword, you'll almost certainly get a syntax error because the compiler will assume that the name refers to a statement. However, because of the possibility that your classes will be accessed by code written in other languages, it is also important that you don't use names that are keywords in other .NET languages. Generally speaking, C++ keywords are similar to C# keywords, so confusion with C++ is unlikely, and those commonly encountered keywords that are unique to Visual C++ tend to start with two underscore characters. As with C#, C++ keywords are spelled in lowercase, so if you hold to the convention of naming your public classes and members with Pascal-style names, they will always have at least one uppercase letter in their names, and there will be no risk of clashes with C++ keywords. However, you are more likely to have problems with Visual Basic, which has many more keywords than C# does, and being non-case-sensitive means that you cannot rely on Pascal-style names for your classes and methods.

Check the Microsoft documentation at `docs.microsoft.com/dotnet/csharp/language-reference/keywords`. Here, you find a long list of C# keywords that you shouldn't use with classes and members.

Use of Properties and Methods

One area that can cause confusion regarding a class is whether a particular quantity should be represented by a property or a method. The rules are not hard and strict, but in general you should use a property if something should look and behave like a variable. (If you're not sure what a property is, see Chapter 3.) This means, among other things, that

- Client code should be able to read its value. Write-only properties are not recommended, so, for example, use a `SetPassword` method, not a write-only `Password` property.
- Reading the value should not take too long. The fact that something is a property usually suggests that reading it will be relatively quick.
- Reading the value should not have any observable and unexpected side effect. Furthermore, setting the value of a property should not have any side effect that is not directly related to the property. Setting the width of a dialog has the obvious effect of changing the appearance of the dialog on the screen. That's fine, because it's obviously related to the property in question.
- It should be possible to set properties in any order. In particular, it is not good practice when setting a property to throw an exception because another related property has not yet been set. For example, to use a class that accesses a database, you need to set `ConnectionString`, `UserName`, and `Password`, and then the author of the class should ensure that the class is implemented such that users can set them in any order.
- Successive reads of a property should give the same result. If the value of a property is likely to change unpredictably, you should code it as a method instead. `Speed`, in a class that monitors the motion of an automobile, is not a good candidate for a property. Use a `GetSpeed` method here; but `Weight` and `EngineSize` are good candidates for properties because they will not change for a given object.

If the item you are coding satisfies all the preceding criteria, it is probably a good candidate for a property. Otherwise, you should use a method.

Use of Fields

The guidelines are pretty simple here. Fields should almost always be private, although in some cases it may be acceptable for constant or read-only fields to be public. Making a field public may hinder your ability to extend or modify the class in the future.

The previous guidelines should give you a foundation of good practices, and you should use them in conjunction with a good object-oriented programming style.

A final helpful note to keep in mind is that Microsoft has been relatively careful about being consistent and has followed its own guidelines when writing the .NET base classes, so a good way to get an intuitive feel for the conventions to follow when writing .NET code is to simply look at the base classes—see how classes, members, and namespaces are named, and how the class hierarchy works. Consistency between the base classes and your classes will facilitate readability and maintainability.

> **NOTE** *The* `ValueTuple` *type contains public fields, whereas the old* `Tuple` *type used properties instead. Microsoft broke a guideline it defined for fields. Because variables of a tuple can be as simple as a variable of an* `int` *and because performance is paramount, it was decided to have public fields for value tuples. No rules without exceptions. Read Chapter 3 for more information on tuples.*

SUMMARY

This chapter examined the basic syntax of C#, covering the areas needed to write simple C# programs. Much of the syntax is instantly recognizable to developers who are familiar with any C-style language (or even JavaScript). C# has its roots with C++, Java, and Pascal (Anders Hejlsberg, the original lead architect of C# was the original author of Turbo Pascal, and also created J++, Microsoft's version of Java).

Over time, some new features have been invented that are also available with other programming languages, and C# also has gotten more enhancements already available with other languages. The next chapter dives into creating different types; differences between classes, structs, and the new records; and an explanation about the members of types such as properties and more about methods.